Test Oracle Automation for V&V of an Autonomous Spacecraft's Planner

Martin S. Feather
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109, USA
Martin.S.Feather@Jpl.Nasa.Gov

Ben Smith
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109, USA
Ben.D.Smith@Jpl.Nasa.Gov

Context

The NASA "Deep Space 1" (DS-1) spacecraft was launched in 1998 to evaluate promising new technologies and instruments. The "Remote Agent", an artificial intelligence based autonomy architecture, was one of these technologies, and in 1999 this software ran on the spacecraft's flight processor and controlled the spacecraft for several days.
We built automation to assist the software testing efforts associated with the Remote Agent experiment. In particular, our focus was upon introducing test oracles into the testing of the planning and scheduling system component. This summary is intended to provide an overview of the work.

Challenges and Opportunities

The Remote Agent experiment used an on-board planner to generate sequences of high-level commands that would control the spacecraft. From a Verification and Validation (V&V) perspective, the crucial observation is that the Remote Agent was to be a self-sufficient autonomous system operating a spacecraft over an extended period, without human intervention or oversight. Hence, V&V of its components, the planner included, was crucial.
Thorough testing was the primary means by which V&V of the planner was performed. However, the nature of the testing effort differed significantly from that of testing more traditional spacecraft control mechanisms. In particular:

- The planner's output (plans) were detailed and voluminous, ranging from 1,000 to 5,000 lines long. Plans were intended to be read by software, and were not designed for easy perusal by humans.
- The planner could be called upon to operate (generate a plan) in a wide range of circumstances. This variety stems from the many possible initial conditions (state of the spacecraft) and the many plausible goals (objectives the plan is to achieve). Thorough V&V required testing the planner on thousands of test cases, yielding a separate plan for each.
- Each plan must satisfy all of the flight rules that characterize correct operation of the spacecraft. (Flight rules may refer to the state of the spacecraft and the activities it performs, and describe temporal conditions required among those states and activities.) The information pertinent to deciding whether or not a plan passes a flight rule was dispersed throughout the plan. This exacerbated the problems of human perusal.

As a consequence, manual inspection of more than a small fragment of plans generated in the course of testing was recognized to be impractical.

The nature of the task – V&V of a software component of an autonomous system – meant that there were significantly increased opportunities for automation. Specifically:

- The data to check was self-contained. Each plan was a self-contained object from which it could be *automatically* determined whether or not each flight rule holds, without need for human intervention.
- The data to check was in a machine-manipulable form, since each plan was intended for consumption by the automatic executive (another software component of the Remote Agent). As a result, it was feasible to develop an automated checker that would work directly on the data available, again, without need for human intervention.
- The conditions to check – the "flight rules" of the spacecraft – were also available in a machine-manipulable form. They were provided as "constraints" to the planner software, expressed in the planner's formal constraint language. As a result, it was feasible to develop a translator that would take as input the flight rules, as expressed for the planner, and *automatically* generate the aforementioned test oracle.

Checking that a plan satisfies constraints is computationally far less complex than finding that plan (planning). Consequently, the checking code is simpler, executes faster, and is easier to develop, than the planner itself – an instance of what Blum terms a "simple checker" (Wasserman and Blum 1997).

Overall Approach

Our approach was to automate the checking of plans. We constructed an automated piece of software that would check the correctness of each plan that resulted from a test run of the planner. In the phraseology of the testing community, this was a "*test oracle*" (Richardson, Aha, and O'Malley 1992). In our case, this correctness checking oracle had two facets:

Verification: our test oracle checked that every flight rule was satisfied by the generated plan. The over 200 such flight rules were automatically checked in this manner, thus verifying that the planner was not generating hazardous command sequences in any of its test runs. Furthermore, this facet of the test oracle was itself generated automatically. We wrote an automatic translator from the planner constraint language into the oracle's checking code.

Validation: our test oracle also performed some validation-like checks. Opportunity for such validation arose from a gap between the "natural" form of some flight rules, and the way they had to be re-encoded so as to be expressed to the planner. We were able to *directly* express the "natural" statements of these flight rules as test oracle checks. Thus, the automatic test oracle was able to help validate that the planner and its inputs were accomplishing the desired behavior.

Further Aspects of Planner V&V

Exploiting Redundancy and Rationale Information During Checking

Each plan contained both a schedule of activities, and a trace relating those actions to the constraints taken into account in their planning. In addition to checking adherence to flight rules, our test oracle also crosschecked this trace information. Specifically, for every constraint that the oracle calculated had a bearing on a planned action, the presence of the corresponding trace information was checked for, and vice-versa.

The advantages of this additional crosschecking were twofold:

- Increased assurance that the test oracle itself was operating correctly.
- Increased assurance that the planner was operating correctly – its plans were correct (adhered to all the flight rules) for "the right reasons". This gave the planner experts more confidence in extrapolating correctly passing the test cases to correctness of the planner in general.

We see this as an instance of the value of crosschecking redundant information, most especially rationale information.

The somewhat redundant-seeming crosschecking made it unlikely that the oracle code itself had a "blind spot" that would overlook a fault in a plan. Eliminating "false positives" (which in our case would correspond to the oracle judging an incorrect plan as correct) is a serious concern (Andrews 1998), and our experience suggests that crosschecking of this kind can help in this regard.

The nature of the trace information being crosschecked is a simple form of rationale – information that justifies *why* an action is there in the plan. Knowledge-based systems can readily provide some form of trace of the reasoning process they follow, yielding such rationale information.

Structuring the Analysis Results

The essential purpose of a test oracle is to determine whether or not a test case satisfies a condition. In our application, we found it beneficial to return more than simply a "pass" or "fail" result for each plan:

- When a plan failed the tests, the oracle returned information indicating which constraint(s) were not met by the plan, and which activities in the plan (or lacking from the plan) were involved. During development of the test oracle generation code, this information was useful to aid the programmer to debug what was wrong with the test oracle; during application of the test oracles in planner testing, this information was useful to aid the planner expert to debug the planner.
- When a plan passed the tests, the oracle reported *which* constraints were exercised by the plan (e.g., only plans that involved engine-thrusting activities were said to have exercised a constraint of the form "every engine-thrusting activity must ..."). The oracle also drew distinctions between several ways in which a constraint could be met. This information was useful for the planning experts to gauge the extent of coverage that the

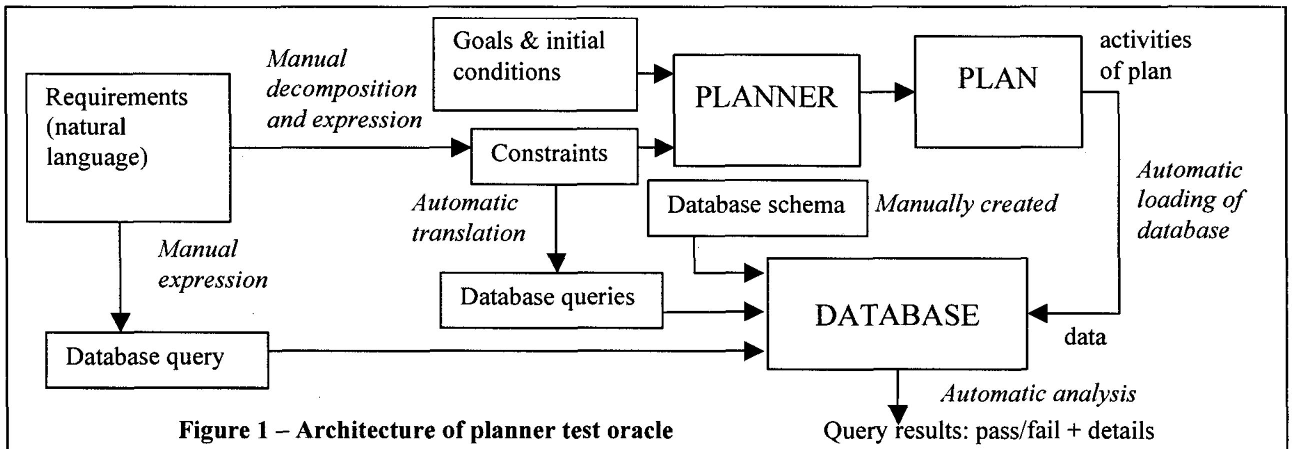

Figure 1 – Architecture of planner test oracle

tests had provided.

Implementation and Development

There are some unusual aspects to the way in which we implemented the test oracles. They used as their underlying reasoning engine a *database*.
The architecture of this is shown in Figure 1. To perform a series of checks of a plan, we automatically loaded the plan as data into the database, having previously created a database schema for the kinds of information held in plans. We expressed the flight rules as database queries. During oracle execution, the database query evaluator was used to automatically evaluate those queries against the data. Query results were organized into those that correspond to passing a test, reported as confirmations, and those that correspond to failing a test, reported as anomalies.
The actual database we used was AP5 (Cohen 1989), a research-quality advanced database tool developed at the University of Southern California. AP5's first-order predicate logic like query language seemed well suited to the expression of the conditions to be checked by the oracle. The constraints input to the planner could be automatically translated into AP5 queries, and the "natural" form of some flight rules could be straightforwardly expressed as further AP5 queries.

Development Process

We followed a staged process to get to the final point of automatically generated test oracles:
- The viability of database-based analysis as a rapid analysis technique was demonstrated in pilot studies on *traditional* design information (Feather 1998).
- A pilot study was used to establish the feasibility of this approach for the checking of the AI planner. The positive results of this pilot study alleviated particular concerns about scalability, and investment of planning experts' time.
- The test oracle generator tool was developed, and deployed during testing of the spacecraft's planner. Development was done primarily by a non-planner expert, aided by a steady stream of advice, information and examples from the planner experts. A planner expert applied the tool during testing, and manually expressed the validation conditions as AP5 queries.

Conclusions

The automatically generated test oracles, augmented by manually expressed validation checks, served to automate what would otherwise have been an impractically burdensome manual activity. Further details of this effort can be found in (Feather and Smith 1999).
Note that this work on test oracles addressed only a small subset of the V&V problems posed by the DS-1 spacecraft's planner. Other significant problems included determining which, and how many, test cases to run, and testing the planner for adequate performance (i.e., generated a plan sufficiently quickly). For further discussion of these issues, the reader is referred to (Smith and Feather 1999), (Smith, Feather, and Muscettola 2000)

Acknowledgements

The research and development described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was provided under NASA's Code Q Software Program Center Initiative UPN #323-098-5b, and by the Autonomy Technology Program. The authors thank the other members of the DS-1 planner team, especially Nicola Muscettola and Kanna Rajan, for their help.

References

Andrews, J.H. 1998. Testing using Log File Analysis: Tools, Methods, and Issues. In Proceedings of the 13th IEEE International Conference on Automated Software Engineering, 157-166, Honolulu, Hawaii.

Cohen, D. 1989. Compiling Complex Database Transition Triggers. In Proceedings of the ACM SIGMOD International Conference on the Management of Data, 225-234, Portland, Oregon.

Feather, M.S. 1998. Rapid Application of Lightweight Formal Methods for Consistency Analysis. *IEEE Transactions on Software Engineering*, 24(11):949-959.

Feather, M.S.; Smith, B. 1999. Automatic Generation of Test Oracles – From Pilot Studies to Application. In Proceedings of the 14th IEEE International Conference on Software Engineering, 63-72, Cocoa Beach, Florida.

Richardson, D. J.; Aha, S. L.; and O'Malley, T. 1992. Specification-based Test Oracles for Reactive Systems. In Proceedings of the 14th International Conference on Software Engineering, 105-118, Melbourne, Australia.

Smith, B.; Feather, M.S. 1999. Verifying an AI Planner for an Autonomous Spacecraft. In Proceedings of the IASTED International Conference on AI and Soft Computing, Honolulu, Hawaii.

Smith, B; Feather, M.S.; Muscettola, N. 2000 Challenges and Methods in Testing the Remote Agent Planner. In Proceedings of the 2nd International Workshop on Planning and Scheduling for Space, San Francisco, California.

Wasserman, H.; and Blum, M. 1997. Software Reliability via Run-Time Result-Checking. *JACM* 44(6):826-845.